ChorusOS 4.0 Production Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 806-3959–10
December 1999

Contents

Tables

Preface

The *ChorusOS 4.0 Production Guide* explains how to use the source code for the ChorusOS™ product to generate an instance of the ChorusOS operating system.

Who Should Use This Book

Use this guide:

- To use a source version of the ChorusOS operating system.
- To see how the Input/Output Manager (IOM) component is built.

This book describes:

- How to perform standard system generation.
- The use of development tools to customize your system.
- Adding source components built with the development tools.
- Configuration of your operating system.

Before You Read This Book

To get the most information from this book you should have already read:

- *ChorusOS 4.0 Introduction*
- *ChorusOS 4.0 Installation Guide for Solaris Hosts*
- *ChorusOS 4.0 Installation Guide for Windows NT Hosts*
- The following man pages: `make`(1S), `m4`(1), `mkmerge`(1CC), `configure`(1CC), `ChorusOSMkMf`(1CC), and `configurator`(1CC).

How This Book Is Organized

Chapter 1 lists the utilities you must have on your system before using the ChorusOS product and also lists the tools, utilities and files you are provided with.

Chapter 2 gives specific information for the installation of ChorusOS source code, and outlines how to generate a ChorusOS system image.

Chapter 3 provides a brief introduction to the `imake` development tool. Further details are available in the *ChorusOS 4.0 Introduction*.

Chapter 4 provides an introduction to the building rules of the `mkmk` tool and a description of the merge method, using `mkmerge`(1CC), which permits the selection of a subset of the source code, for example, the family dependent code.

Chapter 5 describes how to create and add a ChorusOS component to your system, built with the `mkmk` or the `imake` tool. You are guided through an example which is provided in the form of a tutorial.

Chapter 6 supplies information on the customization of your system production by adding features and tunables managed by the `configurator`(1CC) command and the `ews` graphical configuration tool.

Chapter 7 contains details of the XML syntax of the ChorusOS product's configuration files.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks selected product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com[SM] Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	**Prompt**
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	#

CHAPTER 1

Introduction

In the context of this guide, the term *production* denotes all operations that you need to perform in order to generate a bootable system image from the ChorusOS source code provided (the source delivery).

The ChorusOS operating system is composed of components that can be built separately. You have a choice of which components to build. Use the default profiles available in the source files you receive, or customize these files to produce your own operating system.

System Requirements

You must be running the Solaris™ operating environment or using a Windows NT system.

You must have at least 250 megabytes of free disk space to use the source package.

Utilities on Your Host System

In order to install and generate the ChorusOS operating system, your host system must provide the following utilities: `awk`, `basename`, `cat`, `cc`, `chmod`, `cmp`, `cp`, `cut`, `echo`, `egrep`, `find`, `gzip`, `grep`, `head`, `ln`, `ls`, `make`, `m4`, `mkdir`, `mv`, `rm`, `sed`, `sh`, `sort`, `sum`, `tail`, `test`, `touch`, `tr`, `true`, `uname`, `uncompress` and `uniq`. These utilities must be in a directory of your PATH. See “Your Source PATH” on page 22 for further information on PATH.

On Windows NT hosts, these utilities are provided with the Windows Upgrade package which is part of your delivery.

The host file system must support:

- Symbolic links (Solaris operating environment only) or hard links (NT system only).
- Long file names.

Directories

The source files are installed in a default `source` directory that is separate from the work directory.

In this document:

- Your installation directory is referred to as `<install_dir>`. The default installation directory is `/opt/SUNWconn/SEW`.
- Your binary directory is referred to as `<bin_dir>`. The default binary directory is `/opt/SUNWconn/SEW/4.0/chorus-<family>`.
- Your source directory is referred to as `<src_dir>`. The default source directory is `/opt/SUNWconn/SEW/4.0/chorus-<family>/src`.
- Your work directory is referred to as `<work_dir>`.

Components and Tools

The range of components available within the ChorusOS operating system is organized in a hierarchy spanning from the NUCLEUS, at the lowest level, to the EXAMPLES, at the top level as shown in Table 1–1.

TABLE 1–1 Source components in a ChorusOS and their level

component	level
NUCLEUS	nucleus level
DRV DRV_F BSP	board support package level

TABLE 1–1 Source components in a ChorusOS and their level *(continued)*

OS IOM	operating system level
EXAMPLES	applications level

The ChorusOS product ensures compatibility between components built with the two development tools available, `mkmk` and `imake`. The modularity of the source code facilitates porting of the operating system. See the *ChorusOS 4.0 Porting Guide* for more information.

The `mkmk` and `imake` tools provide a mechanism which uses input files in order to create the `Makefile` appropriate to that directory. They:

- provide the rules for compiling
- manage dependencies
- allow you to add components
- enable you to configure source files with `configurator` or the `ews` graphical tool

Introduction to `mkmk`

Table 1–2 lists components built with the `mkmk` tool.

TABLE 1–2 Built with `mkmk`

Component	Description
NUCLEUS	nucleus
OS	POSIX environment
IOM	I/O Manager

The `mkmk` tool is described in this document. It is used to build components at the nucleus and operating system levels (Table 1–2).

Introduction to `imake`

Table 1–3 lists components built with the `imake` tool.

TABLE 1–3 Components Built with `imake`

Component	Description
DRV	Drivers
DRV_F	Family specific drivers
BSP	Boot
EXAMPLES	Applications

For further information on the `imake` tool refer to Chapter 3 of this guide, *ChorusOS 4.0 Introduction* and to the `ChorusOSMkMf`(1CC) man page.

Tools Provided with the ChorusOS System

Table 1–4 gives the list of utilities and production files provided with the ChorusOS system and used with either the `mkmk` or the `imake` development tool. Utilities that can be called directly by developers are described in more detail in the ChorusOS man pages (`configure`, `ChorusOSMkMf`, `configurator` and `mkmerge`). The files in this Table are located in the `<bin_dir>/tools/host/bin` directory.

TABLE 1–4 Host Tools Provided with the ChorusOS System

Utility / File	Description
`configure`	the "configure" script
`cpxml`	wrapper around cp, used to copy XML files
`cproot`	copies files to the target file system
`ChorusOSMkMf`	creates `Makefiles`, using `imake`, for target binaries
`HostMkMf`	creates `Makefiles`, using `imake`, for host binaries
`Makefile.bin`	interface of the tools component
`makedepend`	creates dependencies
`configurator`	handles configuration of features and tunables
`mkimage`	the create system image

TABLE 1–4 Host Tools Provided with the ChorusOS System *(continued)*

`mkmerge`	merges split trees into a merged tree
`genEnv`	reads Makefiles and gives output with their variables
`getExport`	computes the list of object files to use when creating a link
`m4`	the GNU m4 preprocessor
`mkctors`	handles ctors/dtors and tunables during actor link
`mkmake`	wrapper around `make`
`mkmk`	creates `Makefiles`
`mkstubs`	produces system call stubs
`rpcgen`	an RPC protocol compiler
`chgetlayout`	extracts image layout, for DebugServer
`mksymfiles`	ghs compiler only
`getLayoutFile`	ghs compiler only
`concat`	concatenates files
`devsys.mk`	`make` rules to compile host tools

The `imake` files; `Imake.rules`, `Imake.tmpl`, `Package.rules`, `Project.tmpl` are discussed in *ChorusOS 4.0 Introduction*. Table 1–5 gives a description of these files.

TABLE 1–5 The `imake` files provided with the ChorusOS System

File	Description
`imake/Imake.rules`	imake rules
`imake/Imake32.rules`	provides compatibility with r3.2 imake rules
`imake/Imake.tmpl`	template `Makefile`, for `imake`
`imake/Package.rules`	packaging rules, for `imake`
`imake/Project.tmpl`	empty `Project.tmpl`

Each file in the `tgt-make` directory deals with `make` rules for compiling target binary files. Certain files are specific to the `gcc` compiler, others to the `ghs` compiler. Only common and `gcc/powerpc` specific files are described here. All files listed in Table 1–6 are located in `<bin_dir>/tools/tgt-make` directory:

TABLE 1–6 Target Rules for ChorusOS

File	Description
`gcc-devsys.mk`	`make` rules used by the `imake` environment
`gcc-ld.ld`	linker script used to reduce the section number
`gcc-variables.rf`	make variables, for the `mkmk` and `imake` environments
`gcc-tgtdevsys.rf`	`make` rules, for `mkmk`
`variables.rf`	includes the compiler specific file about variables
`tgtdevsys.rf`	includes the compiler specific file about rules
`shared.rf`	calls `mkmake`
`mktgt.rf`	includes all targets rules
`mkrules.rf`	contains rules for the `mkmk` environment
`mkrules.m4`	`m4` macros for `.bf` files
`Makefile.mkimage`	`Makefile` used during image creation
`Makefile.conf`	`Makefile` used during configurable actor links
`genLink`	generic linker, calls `genLink.conf` and `genLink.noconf`
`genLink.conf`	links configurable actors
`genLink.noconf`	links non-configurable actors
`mkdbginfo`	generates offsets and symbol information
`host.conf`	defines the host type
`powerpc/genOff`	create offset files
`powerpc/genOff.awk`	`awk` file used by `genOff`
`powerpc/act.ld`	linker script used for actors using dynamic libraries
`powerpc/lib.ld`	linker script used to create dynamic libraries

Note - None of the above files can be modified.

CHAPTER 2

System Generation

Install the binary components of the ChorusOS 4.0 product before you work with the source delivery. You will use the same method to build the source files. Your installation must include the TOOLS and the EXAMPLES components.

Before you read this Chapter, read

- *ChorusOS 4.0 Installation Guide for Solaris Hosts*, to understand how to install ChorusOS binary files on a Solaris™ operating environment.
- *ChorusOS 4.0 Installation Guide for Windows NT Hosts*, to understand how to install ChorusOS binary files on a Windows NT host.
- The relevant family documentation from the *ChorusOS 4.0 Target Family Documentation Collection*
  - *ChorusOS 4.0 MPC8260 Target Family Guide*
  - *ChorusOS 4.0 MPC8xx Target Family Guide*
  - *ChorusOS 4.0 PowerPC 60x/750 Target Family Guide*
  - *ChorusOS 4.0 UltraSPARC-IIi Target Family Guide*
  - *ChorusOS 4.0 x86/Pentium Target Family Guide.*

Steps to Create a System Image

In this Chapter, the following steps are described.

- Install source components.
- Create a work directory.

- Choose the components to be included in your system image and `configure` them. The `configure` command creates a `Paths` file and a `Makefile` in your work directory.
- Use the `make` command to compile the system image. In your work directory, the `make` command creates `conf`, `obj` and `image` directories and a build directory for each component included in the system image.

The files and directories that you have created during the above steps are discussed in this Chapter.

Installation

Ask your system administrator to supply you with ChorusOS binary and source files. Once you have read the instructions in the ChorusOS 4.0 installation guides, install the source files and the AnswerBook documentation.

Your Source PATH

Depending on your target family, you will use different source code products. Each product has different installation directories. See *ChorusOS 4.0 Target Family Documentation Collection* for further details. The default PATH for a source package is:

```
/opt/SUNWconn/SEW/4.0/chorus-<family>/src
```

where each family is a group of processors, such as PowerPC. This will be your source directory, referred to in this document as <src_dir>.

Note - For your convenience, create an environment variable to shorten your source PATH.

Source File Organization

You receive source packages in read-only mode. The fact that the source files are kept in the source directory, separate from your work directory, has the following advantages

- It prevents the source code from being corrupted during a build.
- It allows you to generate configurations for different targets.
- It permits several users to use the same source code simultaneously.

If you need to modify the source code, use your own copy of the initial source package that you received.

Which Family, Target and Profile

Table 2–1 gives the profile and board dependent components (such as BSP, boot and family specific driver code) to be used with the `configure` command. The PATHs are given relative to <src_dir>.

TABLE 2–1 List of board dependent components

Paths	Component
PowerPC Board	
nucleus/ppc60x/ppc60x	profile
nucleus/bsp/powerpc/genesis2	bsp
nucleus/bsp/powerpc	driver family
MCP750 board	
nucleus/ppc60x/ppc60x	profile
nucleus/bsp/powerpc/mcp750	bsp
nucleus/bsp/powerpc	driver family
SBC8260 board	
nucleus/mpc8260/mpc8260	profile
nucleus/bsp/powerpc/sbc8260	bsp
nucleus/bsp/powerpc	driver family
MPC860 board	
nucleus/mpc860/mpc860	profile
nucleus/bsp/powerpc/mpc8xxADS	bsp
nucleus/bsp/powerpc	driver family
CP1500 board	
nucleus/usparc/usparc	profile
nucleus/bsp/usparc/cp1500	bsp
nucleus/bsp/usparc	driver family
i386AT board	
nucleus/x86/x86	profile

TABLE 2–1 List of board dependent components *(continued)*

nucleus/bsp/x86/i386at	bsp
nucleus/bsp/x86	driver family

All examples given in this document are for the PowerPC family and the genesis2 board. For further information on the family and board that you are using, refer to the *ChorusOS 4.0 Target Family Documentation Collection*.

Which Components

Table 2–2 lists the main source components in a ChorusOS source delivery and the associated directories which are created in your source directory during the installation process.

TABLE 2–2 Source components in a ChorusOS 4.0 delivery and the associated files.

Component	Directory	Description
NUCLEUS	<src_dir>/nucleus	generic kernel code + family profiles
DRV	<src_dir>/nucleus/bsp/drv	generic kernel drivers
DRV_F	<src_dir>/nucleus/bsp/<family>	family specific drivers
BSP	<src_dir>/nucleus/bsp/<family>/<target>	board specific boot code (BSP)
OS	<src_dir>/os	Operating System (network, POSIX)
IOM	<src_dir>/iom	I/O Manager, (drivers, file systems, IP protocol stacks)

System Generation

This section describes the operations necessary to generate a standard system image of the ChorusOS operating system. Standard system generation builds a non-customized version of the operating system.

During the build you use two commands:

- The `configure` command: See the `configure`(1CC) man page for more information.
- The `make` command: See the `make`(1S) man page for more information.

`configure` Command Parameters

You will use three parameters with the `configure` command:

1. Use the -f option to choose the predefined profile for the PowerPC, ppc60x family (Table 2–1), such as;

```
-f <src_dir>/nucleus/sys/ppc60x/ppc60x
```

It selects the NUCLEUS source component which contains a definition of variables for the build.

2. Use the -s option to select the ChorusOS source components to be included in the build of your system image (Table 2–2), such as;

```
-s <src_dir>/nucleus/bsp/powerpc/genesis2 \
      <src_dir>/nucleus/bsp/powerpc \
      <src_dir>/nucleus/bsp/drv
```

The components included determine the identity of the image. In this case, you have included the BSP, DRV and DRV_F components.

3. Use the -b option to include binary components in your system image.

The `configure` command searches the directories specified by the –s and –b options for the `Makefile.src` and `Makefile.bin` files and any file corresponding to `Makefile.*.src` or `Makefile.*.bin`. The command displays each component as it adds it to the configuration. The `configure` command creates two files, `Makefile` and `Paths` in <work_dir>. These files are discussed later in this Chapter.

The make Command

A system image is created from components which you have configured using the configure command. Even though you type a single command, make, the system image is created over four steps, to produce the binary files corresponding to the source components.

1. Selection of the System Image: Choose which of the configured components you will include in your sytem. By changing the components included, you alter the resultant system. The system images are defined in:

   - The Makefile.bin file of the NUCLEUS component:

   ```
   <src_dir>/nucleus/sys/Makefile.bin
   ```

   - The conf directory:

   ```
   <work_dir>/conf/mkimage/mkimage.xml
   ```

   The conf directory is discussed in the next section of this Chapter. Once you have selected your system image, you can use the make command in two ways to build your system image:

   - Referencing the system image name:

   ```
   make <system image_name>
   ```

   - Building all configured components in your work directory:

   ```
   make build
   ```

   The make command automatically performs the next three steps as shown here.

2. Compilation: The default action of make is to compile everything in your work directory, but not to produce a bootable system image. This is because configure may be used to build components which are not directly related to ChorusOS systems.

   Compilation can last from 5 minutes to more than an hour, depending on the host you are using. For the chorus system image, the make build output includes:

   ```
   sh <work_dir>/build-NUCLEUS/mkbuild chorus
   ```

3. Rebuilding the Configurable Components: If the configuration of the system has been changed since the last time the make command was run, using the ews GUI tool or the configurator command, then some actors may need to be relinked. The make command checks if any configurable actors need to be relinked.

```
Running obj/act/pmm/Makefile
       <bin_dir>/tools/host/bin/configurator
       -c /<work_dir>/conf/ChorusOS.xml -action configure
       Running obj/dbg/Makefile
       Running obj/kern/Makefile
       Running obj/os/admin/sys/Makefile
       Running obj/os/am/sys/Makefile
```

```
        Running obj/os/cinit/ftpd/Makefile
        Running obj/os/cinit/hrCtrl/Makefile
        Running obj/os/cinit/rshd/Makefile
        Running obj/os/cinit/teld/Makefile
        Running obj/os/iom/sys/Makefile
```

4. Building of the System Image: Once the necessary actors have been relinked, the `make` command calls `mkimage`. This produces the system image.

```
Start mkimage
        Brief log file: /<work_dir>/image/RAM/chorus/log.brief
        Verbose log file: /<work_dir>/image/RAM/chorus/log.verbose
        Layout file: /<work_dir>/image/RAM/chorus/layout.xml
        Image file: /<work-dir>/chorus.RAM
        Finish mkimage
```

   Once you have built your system image, the work directory contains a build directory for each source component you have included in the build.

Examples of Building a System Image

Here are two examples of how a system image is created using the steps outlined above. It is presumed that you have already installed the binary and source components for your chosen family, as outlined at the beginning of this Chapter.

Example 1: Standard System Generation

Example 1 illustrates standard system generation. The image which you create includes the generic kernel code, PowerPC drivers targeted for the genesis 2 board, generic kernel drivers, and the OS and the IOM components. This is termed a `chorus` system image.

1. Create a work directory:

   host% **`mkdir work_dir`**

2. Change to the work directory:

   host% **`cd work_dir`**

3. Use the `configure` command with a build profile and a selection of components:

   host% **`configure -f <src_dir>/nucleus/sys/ppc60x/ppc60x \`**
   `-s <src_dir>/nucleus/bsp/powerpc/genesis2 \`
   `<src_dir>/nucleus/bsp/powerpc \`

```
<src_dir>/nucleus/bsp/drv \
<src_dir>/os \
<src_dir>/iom
```

4. Create a bootable system image named `chorus`:

   host% **`make chorus`**

   You could also type `make build`, and get the same result, as this system image is already defined in the `Makefile.bin` file of the NUCLEUS component.

Example 2: Kernonly Generation

The system image you create in this Example is the `kernonly` system and contains the generic kernel code, PowerPC drivers, and kernel generic driver component targeted for the genesis 2 board. You create this image in the first steps of porting a system to a new board. See *ChorusOS 4.0 Porting Guide* for more details.

1. Create and change to a work directory as in Example 1.
2. Use the `configure` command:

   host% **`configure -f <src_dir>/nucleus/sys/ppc60x/ppc60x \`**
   `-s <src_dir>/nucleus/bsp/powerpc/genesis2\`
   `<src_dir>/nucleus/bsp/powerpc \`
   `<src_dir>/nucleus/bsp/drv`

3. Create a bootable system image named `kernonly`, using the `make` command:

   host% **`make kernonly`**

Files and Directories Created by `configure` and `make` commands

The `configure` command creates two files, `Makefile` and `Paths`. The `make` command creates a directory for each component included in the system image, as well as a `conf`, `obj` and `image` directory. These files and directories are discussed in this section.

Work Directory Organization

Table 2–3 lists the files and directories generated in your work directory by the `configure` and `make` commands.

TABLE 2–3 Files and Directories generated by the `configure` and `make` commands

Files / Directories	Description
`Makefile`	top level `Makefile`
`Paths`	paths to the source or binary components
`conf/`	system configuration files
`obj/`	object files used by configurable actors
`image/`	image generation files
`build-NUCLEUS/`	build directory for nucleus
`build-DRV/`	build directory for generic drivers
`build-BSP/`	build directory for the board specific boot code
`build-DRV_F/`	build directory for the family specific drivers
`build-OS/`	build directory for the POSIX system
`build-IOM/`	build directory for the I/O manager

Build Directories

The `make` command creates a build directory, in your work directory, for each component included in the system image created. In Table 2–3, there is a build directory for the NUCLEUS (build-NUCLEUS), generic drivers (build-DRV), board specific code (build-BSP), family specific drivers (DRV_F), operating system (build-OS) and I/O manager (build-IOM). Each build directory contains the binary code for the corresponding component.

`Paths` file

The `Paths` file is created in your work directory by the `configure` command. For each source component, the `Paths` file defines two subdirectories, one located in <src_dir> and one in <work_dir>. For example, for the `BSP` component there is a `BSP` directory, which is the component source directory, and `BSP_DIR`, which is the

directory where the component will be generated. For binary components, only one directory, that is the BSP_DIR directory, is defined.

```
...
BSP= <src_dir>/nucleus/bsp/powerpc/genesis2
BSP_DIR= <work_dir>/build-BSP
...
```

The ChorusOS production environment will not modify any file outside of your work directory. This means that regardless of where the source directories of the BSP component are, they will be compiled in the build-BSP subdirectory of your work directory.

Makefile

The Makefile produced by the configure command includes all the Makefiles for each component. View the Makefile.

```
all::DEVTOOLS.all

    PROFILE =  -f <src_dir>/nucleus/sys/ppc60x/ppc60x

    include Paths
    include <src_dir>/nucleus/bsp/drv/src/Makefile.bin
    include <src_dir>/nucleus/bsp/drv/src/Makefile.src
    include <src_dir>/nucleus/bsp/powerpc/Makefile.bin
    include <src_dir>nucleus/bsp/powerpc/Makefile.src
    include <src_dir>/nucleus/bsp/powerpc/genesis2/Makefile.bin
    include <src_dir>/nucleus/bsp/powerpc/genesis2/Makefile.src
    include <src_dir>/os/Makefile.bin
    include <src_dir>/os/Makefile.src
    include <src_dir>/iom/Makefile.bin
    include <src_dir>/iom/Makefile.src
    include <bin_dir>/tools/Makefile.bin
    include <bin_dir>/tools/Makefile.CDS.bin
    include <src_dir>/nucleus/sys/common/Makefile.bin
    include <src_dir>/nucleus/sys/common/Makefile.src
    include <bin_dir>/tools/Makefile.CHSERVER.bin
    include <bin_dir>/tools/Makefile.CHTOOLS.bin

    COMPONENTS =  DRV DRV_F BSP OS IOM DEVTOOLS NUCLEUS  CDS CHSERVER CHTOOLS

    CLEAN =  $(DRV_DIR) $(DRV_F_DIR) $(BSP_DIR) $(OS_DIR) $(IOM_DIR) $(NUCLEUS_DIR)
    clean:; rm -rf $(CLEAN)
    dist: DRV.dist DRV_F.dist BSP.dist OS.dist IOM.dist NUCLEUS.dist

    reconfigure: ; cd /<work_dir>; \
    sh <bin_dir>/tools/host/bin/configure \
    -f /<src_dir>/nucleus/ppc60x/ppc60x \
    -s /<src_dir>/nucleus/bsp/drv /<src_dir>/nucleus/bsp/powerpc \
    /<src_dir>/nucleus/bsp/powerpc/genesis2 /<src_dir>/os /<src_dir>/iom $(NEWCONF)
```

The clean, dist, reconfigure, all and root make targets may be present in Makefiles. They are discussed at the end of this Chapter in relation to the generation of the ChorusOS system image.

In the Makefile shown above, the all target is followed by :: which means you can have multiple update rules. You must use :: if the make command is to work. Note, also, the clean, dist and reconfigure targets in the Makefile shown here.

The top level Makefile of the work directory includes a Makefile.bin and a Makefile.src for each source component. You receive these files with your source delivery. This ensures the compatibility of components even if they are built using different development tools.

Makefile.bin

View the IOM Makefile.bin file for the IOM component, found in the <src_dir>/iom directory.

The output states that the component is the IOM component and gives a list of the components that must be present in the operating system if the IOM component is to work. In this case the OS and NUCLEUS components must be present.

Makefile.src

The Makefile.src file is more complex than the Makefile.bin file, as it describes how the IOM component is compiled. The IOM component is compiled using the mkmk tool.

View the contents of the Makefile.src file for the IOM component, found in the <src_dir>/iom directory.

You are given the information:

- This is an IOM component.
- The IOM.all target is dependent on the DRV, NUCLEUS and OS components and on the DONE file.

This DONE file will be discussed in further detail in the section “Specific Build Options” on page 32.

conf Directory

The conf directory contains files which describe the ChorusOS current configuration. These files are expressed in XML. The details of XML files are provided in Chapter 7.

ChorusOS.xml is the top level configuration file. It contains references to all other configuration files located in the conf directory, as explained in *ChorusOS 4.0 Introduction*. When building the ChorusOS operating system from the source code,

XML configuration files are copied from the various component source files to the `conf` directory.

`obj` Directory

The `obj` directory contains all necessary configurable actors. For instance, the IOM component is compiled in the build-IOM directory. Its object files are copied to the `build-IOM/obj` directory and then linked in the `obj` directory.

`image` Directory

The `image` directory is used during the creation of a system image. The information contained within this directory includes temporary files, log files, symbol tables, and relocated binary files.

Note - If you have created your system image and do not need to manipulate your source delivery further, you will not need to continue reading this document.

Specific Build Options

This section provides further information on directories and files already discussed in this Chapter.

Rebuilding a Component

You can delete any of the directories that were created in your work directory during the generation process. If the directory is necessary, the `make` command will rebuild it. For example, you can remove the BSP component, which in the examples above provides support for the PowerPC board. Then use the `make` command, which will regenerate the component, if it is necessary.

```
host% rm -rf build-BSP/
host% make build
```

The DONE File

When a component is compiled correctly, its Makefile.src file creates a file called DONE, in the build directory. The DONE file exists to prevent make from entering a component's build directory when there is nothing else to compile. If you run the make command, and the Makefile.src file has already created the DONE file, nothing will happen. These DONE files must be removed if a component has been modified, and the dependent components need to be relinked.

Remove the DONE file for the NUCLEUS component:

```
host% rm -f build-NUCLEUS/DONE
```

Run make now and it will enter the NUCLEUS component. Run the make command a second time and you will get no output, as the DONE file has again been created.

Rebuild of the conf Directory

To return to a previous configuration, remove the conf directory, or some files from the conf directory. This returns the system to the default configuration or to the configuration that you updated in the source configuration files. Even if you have not altered the configuration you can rebuild the conf files as follows:

```
host% rm -rf conf/
host% make xml
```

Makefile make Targets

This section details Makefile make targets discussed in the Makefile section above.

- The clean target removes all the build directories from the work directory.
- The dist target produces a binary version of each component. Each component can implement a <component>.dist rule in the component's Makefile.src.
- The reconfigure target adds parameters, using the NEWCONF macro, to the original configure command.
- The all target is the first target in the Makefile and so when the make command is run without an argument make all is run.
- The root target copies files from components into the root directory.

The `all` Target and Component Dependency

Each component implements the `<component>.all` rule, which is defined in the component's `Makefile.bin` file. This rule tells you what other components this component is dependent upon. The `<component>.all` rule in the `Makefile.src` implements the rule, building the component in its own build directory. If this first component depends on a second component, the dependency rule

```
<component1>.all :: <component2>.all
```

is expressed in the `Makefile.bin` file of the first component.

Note - If the dependency is valid only for the build process, the dependency rule is expressed in the `Makefile.src` file.

Update the Source Configuration

Verify that you have installed the EXAMPLES component. This source component, which contains a number of small applications, is built on top of the os and nucleus levels (See Table 1–1 for further details of component hierarchy).

The `reconfigure` target adds components to the initial configuration. Add the EXAMPLES component to your initial work directory, and build it as follows:

```
host% make  reconfigure NEWCONF='-s <src_dir>/opt/examples'
host% make
```

Build the Target NFS root

The `root` target lets you copy files from components into the root directory.

As the EXAMPLES component contains binary files that the target must see, use the `make root` command to copy the binary files of the EXAMPLES component into the root directory as follows:

```
host% make root
```

The `root` directory now contains the binary files of the EXAMPLES component. You can run this component on a target system where this `root` directory can be NFS mounted.

Build a Binary Distribution

To build a binary distribution of the EXAMPLES component, run the following command:

host% **make EXAMPLES.dist**

This command creates a new directory in your work directory, called dist-EXAMPLES, containing the binary files of the EXAMPLES component.

You can move the component's binary files to any directory as follows:

host% **mv dist-EXAMPLES ../**

Use these binary files instead of the source files by using the binary option of the `configure` command.

Use Binary Files Instead of Source Files

To use the EXAMPLES component's binary files, you must remove the current configuration of the component. There is no specific command to achieve this. Follow these four steps:

1. Edit the `reconfigure` rule at the end of the top-level `Makefile` by removing the following line in the source file

```
-s <src_dir>/opt/examples
```

2. Remove the component's build directory and run `make reconfigure` as shown here:

   host% **rm -rf build-EXAMPLES/**
   host% **make reconfigure**

3. Add the application's binary files:

   host% **make reconfigure NEWCONF='-b dist-EXAMPLES'**

4. Run the `make` command and notice the output as `make` rebuilds files in the `conf` directory.

Build in Verbose Mode

Usually, compilations of an `mkmk` or `imake` component are not verbose. If you want to get the whole trace of the build, just add 'SILENT=' to the `Paths` file:

host% **echo 'SILENT=' >> Paths**

To test this modification, compile the EXAMPLES component by configuring the source files:

host% **make reconfigure NEWCONF='-s /<src_dir>/opt/examples/'**

The previous command has produced another `Paths` files, so if you add "SILENT=" again and run the `make` command, you will see a more verbose output.

host% **echo 'SILENT=' >> Paths**
host% **make**

Note - As the application binary files are already configured, the `configure` command will skip the binary files and choose to use the source files.

Build in Debug Mode

To debug a component, you may need to recompile it with special compilation options. To achieve this for the EXAMPLES component:

1. Remove the component's build directory:

   host% **rm -rf build-EXAMPLES**

2. Add 'FREMOTEDEB=ON' to the `Paths` file.
3. Use the `make` command to recompile the component:

   host% **make**

CHAPTER 3

Building with imake

There are two development tools available with the ChorusOS 4.0 product, mkmk and imake. The simpler of these two tools is imake and this tool is recommended to ChorusOS developers for the creation of new ChorusOS components.

Chapter 5 describes how to create your own component with imake.

Four components built with imake are the BSP, the DRV, the DRV_F and the EXAMPLES components.

This document does not describe the imake tool in detail but refers to this tool as a comparison to mkmk.

The imake Tool

The imake tool is not discussed in detail in this document. *ChorusOS 4.0 Introduction* provides information on:

- imake files (Imake.tmpl, Imake.rules, Project.tmpl and Package.rules)
- imake packaging rules
- imake build rules
- examples of how to use the imake tool

The imake files are located in the <bin_dir>/tools/imake directory.

If you are using the imake tool, you do not use the merging operation described in Chapter 4. Instead, imake uses the VPATH variable, which is found in recent versions of make. This means that you can find the source files, regardless of where these directories are located.

Components built with `imake`, such as DRV, DRV_F and BSP, export their public information through packaging rules.

With the `imake` tool you can write applications and adapt your system to a new board by using the boot and driver code provided in the board support package.

`imake` files

An `Imakefile` is a machine-independent description of the targets you want to build. In the first step of the build process, the `imake` tool generates a `Makefile` from each `Imakefile`, by selecting the configuration files with dependencies appropriate to your target system. This has the advantage that the `Imakefile` is a machine-independent description of the targets you want to build and so it is portable.

To produce `Makefiles`, `imake` uses the top level `Project.tmpl` file, and the `Imakefile` contained in each subdirectory. It produces Makefile dependencies which are then written into the `Makefile`.

If files are altered they must be rebuilt. For the `imake` tool within the ChorusOS operating system, only dependencies between source and binary files are taken into account when altered files are rebuilt.

`make` Targets for `imake`

TABLE 3–1 `make` Targets for `imake`

Target	Description
all	default target, build everything
Makefile	rebuild the Makefile in the current directory
Makefiles	rebuild the Makefiles (recursively)
clean	remove produced files (recursively)
depend	generate dependencies (recursively)

CHAPTER 4

Building with mkmk

Read this Chapter only if you want to modify components built with the mkmk tool. Three of these components are the NUCLEUS, the OS and the IOM components.

Note - The mkmk tool is more complex than imake. Use the mkmk tool only if it is mandatory.

This Chapter provides an overview of the mkmk build rules and the utilization of the mkmk tool.

As you receive the IOM component as source files, even in binary deliveries, the examples in this Chapter will refer to building an IOM component, as outlined in Chapter 2.

The mkmk Build Process

The mkmk tool enables you to select a set of files from the source tree, in order to build an operating system for a particular target system. The mkmk tool also provides specific build rules which manage the system configuration through tunables, parameters, and features options.

The mkmk build process can be divided into five steps:

1. Create a link to the source files in the build directory. This operation is called merging and uses the mkmerge utility. The links you have created enable mkmk to locate your source files in <src_dir>.
2. Build the Makefiles in the component's build directory, using mkmk.
3. Build the binary files.

4. Create the dependency files that are used for further builds by the `makedepend` and `getexport` utilities.
5. Build the system images with the `mkimage` command. This phase is described in more detail in the *ChorusOS 4.0 Porting Guide*.

Build Profile

The `build profile` used by the `configure` command is an argument of the `mkmerge` command. There is one default profile for each board, as shown in Table 2–1. For non standard system generation, customize the `build profile` by using the information on the file entries described here.

The `build profile` contains entries of the following type:

- `<var>=<value>`

The range of `<value>` values depends on the variable. Note that:

- `on` and `yes` are equivalent values (both mean “included”)
- `off` and `no` are equivalent values (both mean “not included”)

TABLE 4–1 Entry Type for Build Profile, Defined for PowerPC / ppc60x

merge_dir=<dir>	Specifies the path of the merged tree. `<dir>` will be created if necessary.
tree=<dir1dir2,...>	Specifies directory paths of the split source tree that you want to merge.
target=<ppc>	Specifies the target platform.
debug=<off | on>	The entry is required if the kernel is to be compiled with the DEBUG option set. Default is `off`.
optim=<off | on | size | speed>	Specifies if and how you want to optimize the compilation of your kernel. `size` focuses on size, `speed` on speed while `on` is a good compromise. Setting `optim` to `speed` may make symbolic debugging impossible. Default is `on`.

TABLE 4–1 Entry Type for Build Profile, Defined for PowerPC / ppc60x *(continued)*

profile=<off | on>	Needed if you want to profile your system. Default is `off`.
locks=<rt | gp>	Specifies the implementation of locks within the core executive.

mkmk Files

With the `mkmk` tool, you can write portable description files. The files are independent of both the host system you use and the target for which the system you are creating is destined. The `mkmk` tool also provides high level abstraction rules which ensure the portability of the application build files.

The `mkmk` command uses three types of build description files, suffixed by `.df`, `.bf`, and `.mf`, to produce `Makefiles` and generate `all.dp` dependencies. With `mkmk`, you create a `.mf` file for each host. In this way the `mkmk` tool ensures portability during cross compilation and so ensures the portability of hosts and targets.

.df Files

The `.df` files are used as shell scripts that are launched when the `Makefile` is created to adjust variables before they are written into the `Makefile`. The `.df` files define the following variables:

- The cpp symbols
- The header file directories
- The list of subdirectories

The `.df` files have similar properties to the `Project.tmpl` files of the `imake` tool. See *ChorusOS 4.0 Introduction* for further information. They affect the building of the `Makefiles` for all the subdirectories. With `mkmk`, you can have several `.df` files in each directory of the `source tree`. This contrasts with the `imake` tool, which has only one `Project.tmpl` file per component.

The inheritance mechanism of the `mkmk` tool is different from that of the `imake` tool. With `mkmk`, every `Makefile` is dependent on the `Makefile` present in the parent directory. However, with `imake`, every `Makefile` is dependent on the `Project.tmpl` file present in the top-level of the component and on the few

variables which are inherited from the parent directory. The variable and macros are defined in the production tools. Table 4–2 lists those which you must not modify and Table 4–3 lists those which you may modify.

TABLE 4–2 Variables for the `.df` files, that must not be changed

TARGETMKRULES	`<bin_dir>/tools/tgt-make/mktgt.rf`
MKRULES	`<bin_dir>/tools/tgt-make/mktgt.rf`
DTL	`<bin_dir>/tools/host/bin/`
GROOT	../../
BDIR	`<bin_dir>`
BNAME	kern
HOST	LINUX
FAMILY	PowerPC
COMPILER	gcc
BFILES	List of `.bf` files found in your work directory
MFILES	List of `.lf` files found in your work directory
DFILES	`kern.df`

TABLE 4–3 Variables in the .df files that you can modify

DEFINES	List of macro definitions to use in every compilation rule
INCLUDES	List of directories to search for header files
VARIABLES	The list of user defined variables to export in `Makefiles` produced in subdirectories
SUB_DIRS	The list of directories to use when compiling; these are usually the subdirectories of your work directory
EXTRA_DIRS	A list of directories to use, in addition to `SUB_DIRS`, when creating libraries or linking actors (usually empty)
MODULES	The name of the module to which the object in your work directory (and subdirectories) belong; this is only useful in configurable actors
FEXCEPTION	(ON / OFF); Compile with exception support
FFPU	(ON / OFF): Compile with FPU support
FOPTIM	(ON / OFF): Compile with optimization

TABLE 4–3 Variables in the .df files that you can modify *(continued)*

FPROF	(ON / OFF): Support for profiling
FREMOTEDEB	(ON / OFF): Compile with debug options, to enable debug with XRAY
FVERBOSE	(ON / OFF): Verbose compilations
FWARN	(STRICT / ON / OFF): Enable warnings production during compilation; with STRICT, warnings are treated as errors, the NUCLEUS component uses STRICT

`.bf` Files

The `.bf` files are equivalent to the `Imakefiles`. They define source files, binary files and compilation options. The `.bf` files contain link directives and source definitions, such as `C__SRCS`. You can have several `.bf` files per directory. The `mkmk` tool concatenates the `.bf` files inside the `Makefile` product. The `.bf` file is preprocessed by the macro processor `m4`(1).

Table 4–4 lists the macros used in a `.bf` file. Actor, ConfigurableActor, Library and BigObject, either directly, or indirectly use getExport to get the list of object files. DistFile and Export are defined in `$(DEVTOOLS_DIR)/tgt-make/mkrules.m4`.

TABLE 4–4 Macros used in `.bf` files.

Actor(actor, libs)	build 'actor' using libs
ConfigurableActor(actor, libs)	build 'actor' using libs
Library(lib)	build 'lib'
BigObject(obj)	build a relocatable object file
DistFile(file,dir)	copy 'file' into 'dir'
Export(file,dir)	same as DistFile

`.mf` Files

The `.mf` files contain lists of source files. The `.mf` files are used when the binary file to be built uses many subdirectories, each subdirectory containing a variable number of files to be compiled.

Another form of `.mf` files are `.lf` files. The variables and macros used in the `.mf` and `.lf` files are listed in Table 4–5.

TABLE 4–5 Variables and macros used in `.mf` and `.lf` files

C++SRCS	C++ source files
C__SRCS	C source files
AS_SRCS	assembly code source files
M4_SRCS	M4 source files (assembly code source files preprocessed with m4).
OF_SRCS.	C++ source files used to produce offset files

Merging

The role of `mkmerge` is to link the source files and the component's build directory. For further information, see the `mkmerge`(1CC) man page.

The code for ChorusOS components which contain family specific code is organized in `split trees`. Within the `split trees`, there are subtrees for:

- common code, that is, the code shared by all the platforms
- code that is related to specific families (for example, PowerPC, ppc60x) or CPU

Before generation begins, subtrees are merged into a `merged tree`. There is only one `merged tree` in each merged component. Initially, it contains symbolic, or hard, links that point to files in the `split trees`. The build takes place in the `merged tree`, so the `split trees` remain free of generated files, such as object files and `Makefiles`. You can delete a `merged tree` at any time. This will not affect the source code, as it remains clean in the `split tree`.

From a given set of subtrees of the `split tree`, different `merged trees` can be built. This allows you to produce several system configurations concurrently.

Building the `merged tree` is called the merge operation. When performing this operation, you provide a number of first-level options, called generation options. These options correspond to fundamental production choices, such as the choice of a development system or the target family and are typically found in the build profile given in Table 4–1.

If you are using the `mkmk` tool on a system running the Solaris operating environment, `mkmerge` populates the merge directory with symbolic links. These links point to the source directories required for your build.

If you are using the mkmk tool on a Windows NT system, mkmerge creates hard links between the source directories and your <work_dir>. The use of hard links permits you to use Windows NT hosts to edit files in the <work_dir>.

The Makefiles created by mkmk Tool

During generation, mkmk goes down the build tree and creates a Makefile, for each build directory. The Makefiles are used to build the binary files and each of these Makefiles is autonomous. This means that you can set make running from any directory and all files located in the subdirectories attached to that directory are automatically built. The Makefile content depends on the mkmk production files.

In each directory, where mkmk produces a Makefile, mkmk:

- Calculates the initial value of the variables, in relation to the Makefile of the parent directory.
- Activates the .df script files, located in the current directory, in a parallel shell.
- Asks the shell to recover the values of the variables defined by the .df files.

The list of files to be compiled varies depending on the merge operation. Writing a Makefile that will work on several configurations can be complicated. The solution offered by mkmk is described by the following two steps:

- To place the .mf files found in split trees at the same directory level as the sources files to be compiled.
- To concatenate the .mf files inside the Makefile that it produces.

The .mf files do not have an equivalent within the imake tool.

The mkmk tool will regenerate the Makefile produced, if the .df, .bf or .mf files used to build it change.

Managing Dependencies

Once a file has been modified, for example with an editor, or by a previous compilation, the files that depend on the altered file must be rebuilt. Most development tools only deal with the dependencies between source and binary files. The mkmk system goes further, taking into account any changes which occur in the source files during the build process.

The relationship between the different components is managed through the components' `Makefile.bin` and `Makefile.src` files and through the build paths found in the `Paths` file in your work directory. For example, applications can access both the kernel API and the OS API through the variables set in the `Paths` file:

- `NUCLEUS_DIR` provides the applications with access to the kernel
- `BSP_DIR`, `DRV_DIR`, `DRV_F_DIR` give access to the board support package level information
- `OS_DIR` and `IOM_DIR` enable access to the operating system level API

The information that is accessed between components is the exported information, which is usually present in the binary deliveries.

The IOM Build Directory

To look at a particular example of a built component, list the contents of the build directory for the IOM component, `<work_dir>/build-IOM`. This component's build directory contains five directories, three symbolic links and six files.

TABLE 4–6 Description of the Directories in the IOM Component's Build Directory

Directories	Description
`include`	header files exported by the IOM component
`lib`	libraries used by the IOM component
`obj`	object files needed to link the IOM component
`src`	source files of the IOM component
`conf`	configuration files of the IOM component (used by `mkmerge`)

These five directories will be found in each source component's build directory.

TABLE 4–7 Symbolic Links Between the IOM Build Directory and the Source Files

Symbolic Link	Description
`Makefile.bin`	exports interface with other components
`Makefile.src`	describes how to build the IOM component
`src.df`	build description file

TABLE 4–7 Symbolic Links Between the IOM Build Directory and the Source Files *(continued)*

TABLE 4–8 Files Generated in the IOM Build Directory

File	Description
`DONE`	created by `Makefile.src` when compilation is complete
`SUM`	produced by `Makefile.src`
`profile`	created by the merge process, it contains merge options
`merge.log`	created by the merge process, it contains all `mkmerge` output
`export.lst`	the list of files to export; `mkmake` uses this file to determine which files it should copy and where to copy them to
`Makefile`	the top level `Makefile`, produced by `mkmk`

Examples of IOM Build Files

For the IOM component, the IOM source files are merged into the component's build directory, `<work_dir>/build_IOM`. In this directory, `exports.1st` is produced by the `mkmerge` command. This directory relies on the `profile` file to ensure that the NUCLEUS component is merged before the IOM component.

Look in further detail at the IOM build directory by listing the contents of the `<work_dir>/build-IOM/src/os/iom/sys` directory. This directory contains four files and two symbolic links:

TABLE 4–9 Files Generated in `src/os/iom/sys`

File	Description
`Makefile`	the `Makefile` of the directory, produced by `mkmk`
`N_iom.r`	the IOM actor
`all.dp`	dependency file, produced by `make depend` and `getExport`
`sys_agglo.mk`	internal `Makefile`, used when linking the configurable actor, to copy files from the `src` directory to the `obj` directory

TABLE 4–10 Symbolic links in src/os/iom/sys

Symbolic Link	Description
`sys.bf`	the build file
`sys.df`	the definition file

The contents of the symbolic link files, `sys.bf` and `sys.df`, the `Makefile` and the `/sys/lib/gen/common.mf` file are given here.

sys.df

The `sys.df` file, defines the INCLUDES and DEFINES variables.

```
INCLUDES="    \
     -Isys/include   \
     -Ibsd    \
     -Ibsd/sys   \
     -I${OS_DIR}/include/sys \
     -Ibsd/machine   \
     -I${OS_DIR}/include/machine \
     -I${OS_DIR}/include/chorus/iom \
     -I${OS_DIR}/include/chorus \
     -I${OS_DIR}/include/chorus/cx \
     -I${NUCLEUS_DIR}/include/chorus \
     -I${IOM_DIR}/include  \
     -I${OS_DIR}/include  \
     -I${NUCLEUS_DIR}/include \
     -I${OS_DIR}/include/stdc \
     -I${NUCLEUS_DIR}/include/stdc"

    DEFINES="$DEFINES -DKERNEL  -D_KERNEL -D__FreeBSD__ /
    -DINET -DNO_CACHE -DMSDOSFS -DNFS -DIOM_MALLOC  -DSHARED_FD"
```

sys.bf

In this example, `S_LIBS` contains the list of libraries used to link the IOM component. The `ConfigurableActor` rule is also used to link the IOM component.

```
S_LIBS = $(IOM_DIR)/lib/os/iom/sys/lib/ufs.a \
      $(IOM_DIR)/lib/os/iom/sys/lib/disk.a \
      $(IOM_DIR)/lib/os/iom/sys/lib/gen.a \
      $(IOM_DIR)/lib/os/iom/sys/lib/mem.a \
      $(IOM_DIR)/lib/os/iom/sys/lib/unresolved.a \
      $(NUCLEUS_DIR)/lib/stdc/libC.a \
      $(NUCLEUS_DIR)/lib/embedded/libebd.s.a \
      $(NUCLEUS_DIR)/lib/cpu/cpu.s.a \
      $(OS_DIR)/lib/classix/libcx.a \
      $(NUCLEUS_DIR)/lib/classix/libsys.s.a
```

```
ConfigurableActor(N_iom.r, $(S_LIBS))
```

Makefile

The `Makefile` generated by `mkmk` contains two parts. View the contents of the `Makefile`. The first part has a list of the definitions of variables as defined in Table 4–2, Table 4–3, Table 4–4 and Table 4–5. The second part contains preprocessed copies of the `.bf`, `.lf`, and `.mf` files found in the work directory. In this example there is only a `.bf` file present in the work directory.

```
#
#  This makefile is generated automatically
#   in build-IOM/src/os/iom/sys
#
...
BDIR =  <work_dir>
BNAME = sys
MPATH = build-IOM/src/os/iom/sys
DEFINES = -DNDEBUG -DKERNEL -D_KERNEL -D__FreeBSD__ -DINET \
-DNO_CACHE -DMSDOSFS -DNFS -DIOM_MALLOC -DSHARED_FD
INCLUDES = -Isys/include -Ibsd -Ibsd/sys
-I/<work_dir>/build-OS/include/sys -Ibsd/machine \
-I/<work_dir>/build-OS/include/machine \
-I/<work_dir>/build-OS/include/chorus/iom \
...
FEXCEPTION = OFF
FFPU = ON
FOPTIM = ON
FPROF = OFF
...
FAMILY = ppc60x
COMPILER = gcc
VARIABLES = OS_DIR IOM_DIR NUCLEUS_DIR DRV_DIR
SUB_DIRS = bsd sys
EXTRA_DIRS =
BFILES = sys.bf
MFILES =
DFILES = sys.df
MODULES =
OS_DIR = <work_dir>/build-OS
IOM_DIR = <work_dir>/build-IOM
NUCLEUS_DIR = <work_dir>/build-NUCLEUS
DRV_DIR = <work_dir>/build-DRV

include $(BDIR)/Paths

include $(DEVTOOLS_DIR)/tgt-make/shared.rf

# produced from sys.bf -- begin
 S_LIBS = $(IOM_DIR)/lib/os/iom/sys/lib/ufs.a \
          $(IOM_DIR)/lib/os/iom/sys/lib/disk.a \
          $(IOM_DIR)/lib/os/iom/sys/lib/gen.a \
          $(IOM_DIR)/lib/os/iom/sys/lib/mem.a \
          $(IOM_DIR)/lib/os/iom/sys/lib/unresolved.a \
          $(NUCLEUS_DIR)/lib/stdc/libC.a \
```

```
            $(NUCLEUS_DIR)/lib/embedded/libebd.s.a \
            $(NUCLEUS_DIR)/lib/cpu/cpu.s.a \
            $(OS_DIR)/lib/classix/libcx.a \
            $(NUCLEUS_DIR)/lib/classix/libsys.s.a

     BINARY += N_iom.r

     N_iom.r: $(S_LIBS)
             $(MKLINK) -r -c -e _start -o N_iom.r -B $(S_LIBS)
    # produced from sys.bf -- end

    include all.dp
    include $(DEVTOOLS_DIR)/tgt-make/mktgt.rf
```

common.mf File

To examine a directory that has a `.mf` file, go to the `<work_dir>/build-IOM/src/os/iom/sys/sys/lib/gen` directory. List the contents of the `common.mf` file:

```
...
    C__SRCS = \
            insremque.c ovbcopy.c memstat.c iomRqTask.c util.c
```

The `.mf` files contain the definition of variables and nothing else. These variables are listed in Table 4–5.

all.dp Dependency File

The `Makefile` listed above for the IOM component, located at `<work_dir>/build-IOM/src/os/iom/sys`, includes a dependency file, `all.dp`. This dependency file is located in the same directory as the `Makefile`. In the IOM component, the `all.dp` file contains the list of files upon which the linked binary files, such as actors or libraries, are dependent. The `-export` lines at the top of the file are used by `getExport` during the link phase. In this example, as the directory contains no object file, these lines are empty.

host% **head all.dp**

```
 # automatically generated file
    # -export ALL :
    # -export SUP :
    # -export USR :
    N_iom.r: bsd/dev/conf/all.dp
    N_iom.r: bsd/dev/conf/conf.o
    N_iom.r: bsd/dev/console/all.dp
    N_iom.r: bsd/dev/console/console.o
    N_iom.r: bsd/dev/flash/all.dp
    N_iom.r: bsd/dev/flash/flashdrv.o
    ...
    N_iom.r: all.dp
```

Building an IOM Component

The `make` process occurs in three phases

1. Build source files (this phase is only present in the NUCLEUS component).
2. Compile source files and create dependency files.
3. Link object files.

When `make` runs, it launches `mkmake`. The `mkmake` tool uses the `all.dp` dependency file to determine in which directories `make` will run. Only the directories accessed are displayed.

```
host% make
...
<bin_dir>/tools/host/bin/mkmake
<work_dir>/build-NUCLEUS/ sources +l +b -- sources
>> In src/os/iom/sys sources
<< In src/os/iom/sys sources done
<bin_dir>/tools/host/bin/mkmake
<work_dir>/build-NUCLEUS/ prod +l +m +b -- local_prod
>> In src/os/iom/sys prod
<< In src/os/iom/sys prod done
<bin_dir>/tools/host/bin/mkmake
<work_dir>/build-NUCLEUS/ link +b -- local_link
>> In src/os/iom/sys/bsd/kern/disk link
<< In src/os/iom/sys/bsd/kern/disk link done
...
```

Relink of the IOM Actor

To relink the IOM actor, remove it and then run the `make` command to regenerate the component.

```
host% rm <work_dir>/build-IOM/src/os/iom/sys/N_iom.r
host% make
```

The output you get displays the regeneration of the `N_iom.r` actor.

Recompilation of a Source File

Recompile a source file, for example `pathName.C`, by removing the object file, `<work_dir>/build-IOM/src/os/iom/sys/sys/lib/mem/pathName.o` and running the `make` command. You get a very long output which includes the following:

```
host% rm <work_dir>/build-IOM/src/os/iom/sys/sys/lib/mem/pathName.o
host% make
```

```
>> In src/os/iom/sys/sys/lib/mem prod
     ...
     CC pathName.C
     ...
     ar -> mem.a
     ...
     sh <bin_dir>/tools/host/bin//../../tgt-make/genLink \
     <work_dir>/build-NUCLEUS/ <work_dir> -r -c -e _start -o N_iom.r -B \
     ...<work_dir>/build-NUCLEUS/lib/classix/libsys.s.a

     <bin_dir>/tools/host/bin/configurator \
     -c <work_dir>/conf/ChorusOS.xml -action configure

     <bin_dir>/tools/powerpc/solaris/5.00/powerpc-elf/bin/gcc \
     ...
     bsd/kern/vfs/vfs_vnops.o \
     ...
     <bin_dir>/tools/host/bin/mkctors -T <work_dir>/obj/os/iom/sys/tunables.k \
     <work_dir>/obj/os/iom/sys/N_iom.r.xpO > <work_dir>/obj/os/iom/sys/N_iom.r.CT.s
     <bin_dir>/tools/powerpc/solaris/5.00/powerpc-elf/bin/gcc
     ...
     <work_dir>/obj/os/iom/sys/N_iom.r.CT.o
     ...
     <bin_dir>/tools/powerpc/solaris/5.00/powerpc-elf/bin/gcc
     ...
     <work_dir>/obj/os/iom/sys/N_iom.r.CT.o
     ...
```

Look at this output as occurring in several steps.

1. For step 1:

```
...
    >> In src/os/iom/sys/sys/lib/mem prod
    ...
    CC pathName.C
    ...
    ar -> mem.a
    ...
```

   The <work_dir>/build-IOM/src/os/iom/sys/sys/lib/mem directory builds a library used during the IOM link. The object file is generated, the library is updated, and the mkmake command continues.

2. For step 2:

```
...
   sh <bin_dir>/tools/host/bin//../../tgt-make/genLink \
   <work_dir>/build-NUCLEUS/ <work_dir> -r -c -e _start -o N_iom.r -B \
   ...<work_dir>/build-NUCLEUS/lib/classix/libsys.s.a
```

   The linked actor is copied back to the source directory.

3. Step 3 calls the configurator command:

```
 <bin_dir>/tools/host/bin/configurator \
    -c <work_dir>/conf/ChorusOS.xml -action configure
```

This gets the IOM configuration settings from the XML configuration file.

4. The last step is the link itself.

```
<bin_dir>/tools/powerpc/solaris/5.00/powerpc-elf/bin/gcc    ...
   bsd/kern/vfs/vfs_vnops.o \...
   <bin_dir>/tools/host/bin/mkctors -T <work_dir>/obj/os/iom/sys/tunables.k \
   <work_dir>/obj/os/iom/sys/N_iom.r.xpO > <work_dir>/obj/os/iom/sys/N_iom.r.CT.s
   <bin_dir>/tools/powerpc/solaris/5.00/powerpc-elf/bin/gcc
   ...
   <work_dir>/obj/os/iom/sys/N_iom.r.CT.o
   ...
   <bin_dir>/tools/powerpc/solaris/5.00/powerpc-elf/bin/gcc
   ...
   <work_dir>/obj/os/iom/sys/N_iom.r.CT.o
   ...
```

This is done after several loops of `gcc`, and a call to `mkctors`, in order to set the values of the `tunable` parameters.

Link of Configurable Actors

Steps 2 to 4 comprise the link of a configurable actor; in this instance, `N_iom.r`. The link of a configurable actor is unusual, as instead of linking the actor in the current work directory, object files are copied into the `obj` directory, located in `build-IOM`. The actor is linked to your work directory and then copied back to the source tree.

The first step is to call `genLink`, which will:

- Produce the `sys_agglo.mk` Makefile, as seen in the IOM component above.
- Copy object files to the `obj` directory.
- Generate two `Makefiles` in the `obj` directory. These `Makefiles` will perform the link.
- Call the two `Makefiles`.
- Copy the linked actor back to the source directory.

Verbose Mode

See how the `mkmake` command is called for each phase by running the `make` command in verbose mode as follows:

```
host% make SILENT=
 /<bin_dir>/tools/host/bin/mkmake \
   /<work_dir>/build-NUCLEUS/ sources +l +b -- sources
   >> In <bin_dir>/bin/tools/host/bin/mkmake \
```

```
<work_dir>/build-NUCLEUS/ prod +l +m +b -- local_prod
<bin_dir>/tools/host/bin/mkmake \
<work_dir>/build-NUCLEUS/ link +b -- local_link
>> In src/os/iom/sys/sys/lib/mem link
<< In src/os/iom/sys/sys/lib/mem link done
```

depend and all

The make depend command rebuilds dependencies. This make target is used when the list of header files in a source file is changed. As seen earlier, if the Makefile content changes, dependencies are reconstructed. In the <work_dir>/build-IOM/src/os/iom/sys/sys/lib/gen directory, use the make depend command.

```
host% make depend
     >> In src/os/iom/sys/sys/lib/gen depend
    ...
    Makefile -> all.dp
    << In src/os/iom/sys/sys/lib/gen depend done
```

Note - In this directory, there are object files that are used in linking the IOM component. In this case, the -export lines in all.dp are not empty.

```
# automatically generated file
    # -export ALL :   insremque.o ovbcopy.o memstat.o iomRqTask.o util.o
    # -export SUP :   insremque.o ovbcopy.o memstat.o iomRqTask.o util.o
    # -export USR :   insremque.o ovbcopy.o memstat.o iomRqTask.o util.o
```

This information is used by getExport, which reads all the all.dp files in the source tree to determine which object files to use.

make Targets for mkmk

The complete list of make targets that you can use with the mkmk tool are shown here. The section above shows depend and all make targets.

TABLE 4–11 Description of make Targets used with mkmk

Target	Description
all	default target, builds everything
Makefile	rebuild the Makefile in current directory

TABLE 4–11 Description of `make` Targets used with `mkmk` *(continued)*

makemk	rebuild the Makefiles (recursively)
merge	remerge the current component
makesrc	build offset files (recursively)
sources	build offset files in the current directory
makeprod	build object files and libraries (recursively)
prod	build object files and libraries in the current directory
makelink	link libraries and actors (recursively)
link	link libraries and actors in the current directory
depend	generate dependencies (recursively)
clean	remove files produced (recursively), with the exception of `Makefiles` and dependency files
clobber	remove files produced (recursively)

`make mkmk`

By using the `make mkmk` command, the `Makefiles` can be built or rebuilt.

CHAPTER 5

Creating a ChorusOS Component

This Chapter shows you how to create your own component for your ChorusOS system. You can create this type of component with `mkmk`, `imake` or a tool of your choice.

Introduction

A component must provide a set of `Makefiles` in order to be configured using the `configure` tool. These `Makefiles` are as follows:

- `Makefile.bin` for binary components
- `Makefile.src` and `Makefile.bin` for source components.

The `Makefile.src` file is used to build a component. The component can be built using any build tool, such as the `mkmk` tool or the `imake` tool.

When you receive the ChorusOS product, the `Makefile.src` and `Makefile.bin` files are provided, for each component. This has the advantage that, prior to configuration, all the components have characteristics in common. This means that the components are compatible, even when they have been built using different tools.

You can choose a selection of components generated with the `mkmk` and `imake` tools from the lists given in Chapter 1 in order to create your system image.

You can also create your own component by creating the `Makefile.bin` and `Makefile.src` files yourself as shown in this Chapter.

mkmk Component

This section describes how to build a ChorusOS component with mkmk and the basic operations needed to manage your component. The information is provided in the form of a tutorial, leading you through an example which shows you how to achieve each step.

Creating a Component

To build a very simple ChorusOS component called MYCOMP, which contains an application that displays a short message, you must begin by creating a path in your /tmp directory which contains your new component.

host% **cd /tmp/MYSRC**
host% **mkdir MYCOMP**

Makefile.bin

Create a Makefile.bin file, containing the following information, in your MYCOMP directory.

```
COMPONENT += MYCOMP
   ROOT      += $(MYCOMP_DIR)/root
```

The first line in the output, which declares the component's name, is mandatory. As the COMPONENT variable is the list of all components to be configured, use '+=', and not '='. The ROOT variable contains a list of directories to be copied to the target root file system.

Makefile.src

Create a Makefile.src, containing the following information, in your MYCOMP directory.

```
all:: MYCOMP.all

    MYCOMP.all:: NUCLEUS.all OS.all
    MYCOMP.all:: $(MYCOMP_DIR)/DONE

    $(MYCOMP_DIR)/DONE:
     rm -rf $(MYCOMP_DIR)
     $(DEVTOOLS_DIR)/host/bin/mkmerge -s $(MYCOMP) -t $(MYCOMP_DIR)
     cd $(MYCOMP_DIR); $(DEVTOOLS_DIR)/host/bin/mkmk -t $(NUCLEUS_DIR)
     cd $(MYCOMP_DIR); $(make)
     touch $(MYCOMP_DIR)/DONE
```

This file is more complex than the `Makefile.bin` file as it describes how to build the component. The first lines of output give the list of components that must be built before the MYCOMP component. As the application that you are building needs operating system services, you `must` build the NUCLEUS and OS components before building the MYCOMP component.

The other lines in the `Makefile.src` file explain the following phases of the build of an `mkmk` component:

- Creation of the merged tree with `mkmerge`. The `MYCOMP` variable points to the source directory of the component, and `MYCOMP_DIR` points to the component's build directory. Both variables are provided by the `Paths` file. This file is located at the top of the working directory and is initialized by the `configure` command.
- Creation of the `Makefiles` file with `mkmk`.
- Running of the `make` command in the merged tree. It is important to use $(`make`) instead of $(`MAKE`), as the `make -n` option executes lines containing $(`MAKE`) instead of displaying the file.
- Declaration that the component compilation is complete.

Adding the Component to the System Configuration

Even though the component, MYCOMP, is empty, it may be configured. Use the `make reconfigure` command in your work directory, that is to say the directory in which you have previously run `configure` to build your system image. To compile your component type `make`:

host% **`make reconfigure NEWCONF='-s /tmp/MYSRC/MYCOMP'`**

Type `make` to compile your component and you will see the following output.

```
...
rm -rf /<work_dir>/build-MYCOMP
/<work_dir>/build-DEVTOOLS/host/bin/mkmerge -s /tmp/MYSRC/MYCOMP \
-t /<work_dir>/build-MYCOMP
merged tree installed in /<work_dir>/build-MYCOMP
cd /<work_dir>/build-MYCOMP; /<work_dir>/build-DEVTOOLS/host/bin/mkmk \
-t /<work_dir>/build-NUCLEUS
>> In build-MYCOMP makemk
<< In build-MYCOMP makemk done
cd /<work_dir>/build-MYCOMP; make
touch /<work_dir>/build-MYCOMP/DONE
```

Creating a Simple Hello Application

Create a subdirectory, `src`, within your `MYCOMP` directory.

To create a simple hello application,

1. create the following three files in the `MYCOMP/src` directory:

   - `hello.c`, this source file will say hello and is written as follows:

```
#include <stdio.h>
       extern void bye();
       main()
       {
           printf("Hello\n");
       #ifdef BYE
           bye();
       #endif
           exit(0);
       }
```

   - `hello.bf`, this file provides rules to build the application and contains the following information:

```
C__SRCS = hello.c
      LIBS = $(OS_DIR)/lib/classix/libcx.a $(NUCLEUS_DIR)/lib/classix/libsys.s.a
      Actor(hello, $(LIBS))
      Export(hello, $(MYCOMP_DIR)/root/bin)
```

     The `hello.bf` file declares:

     - The source files to be compiled.
     - The libraries to be used.
     - The fact that you want to build the `hello` actor.
     - The fact that you want to copy the `hello` actor into the `root/bin` directory, once it is built.

   - `hello.df`, this file defines the variables and contains the following information:

```
../../Paths
  VARIABLES="OS_DIR NUCLEUS_DIR"
  BDIR=${BUILD_DIR}
  INCLUDES="-I${OS_DIR}/include -I${NUCLEUS_DIR}/include \
  -I${OS_DIR}/include/chorus -I${NUCLEUS_DIR}/include/chorus"
```

     The `hello.df` file contains two parts. The first three lines give the location of the various components that MYCOMP depends on. The `INCLUDES` variable shows where to look for header files.

2. Build your component in your work directory using the `make` command:

   host% **`make`**

3. Add the `hello` binary file to the target file system, using the following command:

   host% **`make root`**

Updating your Application with Source Files in Several Directories

Applications may have several source files in different directories. These files may be accessed using links. This example shows how to create a link between the files.

Define the BYE flag by creating a `bye.df` file in your `MYCOMP/src`, including the following information:

```
DEFINES="-DBYE"
```

The `bye()` function will be called once the BYE flag is defined in the compilation options.

Remove the `MYCOMP` build directory and run `make` again as follows:

```
host% rm -rf /<work_dir>/build-MYCOMP
host% make
```

The link phase fails because the `bye()` function in not defined properly.

Create a new directory (`bye`) in your `src` directory and define the `bye()` function in the `bye.c` source file by including the following information in the file:

```
void bye() {}
```

The `bye.c` file is not in the same directory as the `.df` file. You must create a `.mf` file, `bye.mf`, in your `bye` directory containing the following information:

```
C__SRCS = bye.c
```

Run `make` again and the link phase will succeed.

```
host% make
```

Note, the `bye.o` object file has been automatically added to the list of files used to link `hello`. To display this list, use the `getExport` command:

```
host% <bin_dir>/host/bin/getExport build-MYCOMP/src
build-MYCOMP/src/bye/bye.o
    build-MYCOMP/src/hello.o
```

Using Merge to Update your Build Directory

If links produced by `mkmerge` are removed, `make merge`, run in the merged tree, will recreate them:

```
host% cd build-MYCOMP/src
host% ls
Makefile  bye     hello     hello.c hello.o
all.dp   bye.df    hello.bf  hello.df
host% rm -rf bye
host% make merge
/<work_dir>/build-MYCOMP/src/bye/bye.c -> \
    tmp/MYSRC/MYCOMP/src/bye/bye.c
    /<work_dir>/build-MYCOMP/src/bye/bye.mf -> \
    tmp/MYSRC/MYCOMP/src/bye/bye.mf
    merged tree installed in /<work_dir>/build-MYCOMP
```

The merge process in the ChorusOS system, is controlled by `merge.rf` files. These files contain commands that are executed by `mkmerge`. The syntax and semantics of these files are described in the `mkmerge` man page.

To have the `bye` subdirectory of the split tree appear as `ciao` in the merge tree, use a `merge.rf` file in the `bye` source directory as follows:

```
host% echo 'move ../ciao' > /tmp/MYSRC/MYCOMP/src/bye/merge.rf
host% rm -rf bye
```

Run the `merge` command again and you will see that you have created a new subdirectory.

```
host% make merge
```

Rebuild the `Makefiles` using the `make makemk` command:

```
host% make makemk
>> In build-MYCOMP/src makemk
>> In build-MYCOMP/src/ciao makemk
<< In build-MYCOMP/src/ciao makemk done
<< In build-MYCOMP/src makemk done
...
```

Creating a Library

You can create a library with `mkmk` by following this example. Create a `.bf` file, `bye.bf`, that calls the `Library` macro, putting the following information:

```
Library(ciao.a)
```

Type `make merge`.

```
host% make merge
/<work_dir>/build-MYCOMP/src/ciao/bye.bf -> \
    /tmp/MYSRC/MYCOMP/src/bye/bye.bf
    merged tree installed in /tmp/WORK/build-MYCOMP
```

This library will automatically include the list of object files given by `getExport`.

Linking your Application to the Library

To link the `hello` actor to the new `ciao.a` library, the `hello.bf` file, in your `MYCOMP/src` directory, must be modified to contain the following information:

```
C__SRCS = hello.c
   LIBS = ciao/ciao.a $(OS_DIR)/lib/classix/libcx.a \
   $(NUCLEUS_DIR)/lib/classix/libsys.s.a
   Actor(hello, $(LIBS))
   Export(hello, $(MYCOMP_DIR)/root/bin)
```

Rebuild the `Makefiles` and the `hello` actor as follows:

```
host% make makemk
host% make
 ...
 cc bye.c
 Makefile -> all.dp
 ar -> ciao.a
 ...
 sh /<work_dir>/build-DEVTOOLS/host/bin//../../tgt-make/genLink \
 /<work_dir>build-NUCLEUS/ /<work_dir> -r -e _start -o hello -B ciao/ciao.a \
 /<work_dir>/build-OS/lib/classix/libcx.a \
 /<work_dir>/build-NUCLEUS/lib/classix/libsys.s.a
 ld -> hello.xpO
 hello.xpO -> hello.CT.o
 export hello
 << In src link done
```

Rebuilding a `Makefile`

You can have applications with multiple `.mf` files in your directory, particularly when using the `mkmerge` command to integrate specific or optional code. When a `.mf` file is created or changed, you must recreate the `Makefile`. To check this, create an empty directory, and put a `.mf` file, `a.mf`, in it. Type either `make Makefiles` or `make mkmake` and you will recreate the `Makefile`.

```
>> In <work_dir>/build-IOM/src/os/sys/sys/lib/gen makemk
<< In <work_dir>/build-IOM/src/os/iom/sys/sys/lib/gen makemk done
```

If the `Makefile` changes, the compilation options may have changed. Therefore all object files must be recompiled. When `mkmake` regenerates a `Makefile`, if the `Makefile` has changed, `mkmake` removes any object files which have been produced. Use the `make` command now to recompile the source code.

imake Component

For imake components, the Imakefile is used to create the Makefile. The imake Makefile.src is similar to the Makefile.src file described for mkmk. If the MYCOMP component is built using the imake tool, its Makefile.src would be:

```
all:: MYCOMP.all

MYCOMP.all:: DEVTOOLS.all NUCLEUS.all OS.all
MYCOMP.all:: $(MYCOMP_DIR)/DONE

$(MYCOMP_DIR)/DONE:
 rm -rf $(MYCOMP_DIR)
 sh $(DEVTOOLS_DIR)/ChorusOSMkMf $(BUILD_DIR) \
 -s $(MYCOMP) -b $(MYCOMP_DIR) -d $(MYCOMP_DIR)
 cd $(MYCOMP_DIR); $(make) Makefiles
 cd $(MYCOMP_DIR); $(make)
 touch $(MYCOMP_DIR)/DONE
```

The Makefile.src contains the call to ChorusOSMkMf which will generate all Makefiles in the component. Then, make Makefile and make are called.

As the binary components have already been created, the Makefile.bin file of a component does not depend on the component's generation method. There is no difference between a Makefile.bin file for an imake component and a Makefile.bin file for a mkmk component.

The contents of the Makefile.bin are as follows.

```
COMPONENT += MYCOMP
   ROOT      += $(MYCOMP_DIR)/root
```

ChorusOS 4.0 Introduction explains how to create a hello application with imake.

Other Components

Components that do not use either the mkmk or the imake tool are free to use any build method in the $(MYCOMP_DIR)/DONE rule. There is a restriction that only one directory, $(MYCOMP_DIR) (in this example build-MYCOMP) can be considered as writable. If the build method produces binary files in source directories, the source files should be projected from $(MYCOMP) to $(MYCOMP_DIR) with mkmerge. Typing the following command.

```
<bin_dir>/host/bin/mkmerge -s $(MYCOMP) -t $(MYCOMP_DIR)
```

This command will populate `MYCOMP_DIR` with links that point back to your `MYCOMP` directory.

CHAPTER 6

Customization

ChorusOS Configuration

ChorusOS 4.0 Introduction explains how to configure the ChorusOS operating system with the `configurator` command, or by using the `ews` graphical tool. With both configuration tools, you can statically configure the features, tunables and environment variables. Once the system configuration has been altered, rebuild a new system image.

Note - It is strongly advised that you use the `ews` graphical tool, particularly if you are not familiar with the `configurator` command, as the `ews` graphical tool ensures the ChorusOS system integrity.

This chapter explains how to implement the features and tunables managed by the `mkmk` tool.

- Features control the list of modules linked to build an actor. The list of object files used to link an actor is calculated from the values of the features values.
- Tunables provide a way of changing the values of integers used in the actor. There is no recompilation of the source files. Values are taken into account when the actor is linked.

Only components built with the `mkmk` tool can implement features and tunables. Features and tunables apply to configurable actors in which the ChorusOS kernel itself is included.

Configuration files are all expressed in eXtensible Markup Language (XML). This provides a structured and organized view of the various configuration options of the ChorusOS system. See *ChorusOS 4.0 Introduction* for a description of the XML configuration language, as used in the ChorusOS system.

Adding a Tunable

To add a tunable (`my_tunable`) to the IOM component, an integer named `my_tunable` needs to be written in an IOM source file, as follows:

```
extern int my_tunable;
```

You will configure this integer using the external name `iom.my.tunable`. The default value for `my_tunable` is 0.

Execute the following steps:

1. To add the tunable to the XML source configuration file for the IOM component, go to the directory containing the IOM configuration files:

   host% **`cd <src_dir>/iom/conf/mkconfig`**

   To add the `iom.my.tunable`, modify two files, `iom_rule.xml` and `iom_action.xml`.

   - The `iom_rule.xml` file contains the description of the configurable entities as features and tunables, and their associated dependency.
   - The `iom_action.xml` file contains the internal implementation rules for management of features and tunables and provides the interface with the `mkmk` environment.

   Include the definition of the tunable in XML in the `iom_rule.xml` file.

```
<tunable name="iom.my.tunable">
        <description> My Tunable </description>
        <int>
        <const>0</const>
        </tunable>
```

   This definition includes the external name, a description field that will be accessible through the `ews` configuration tool, and a default value. All tunable values are integers; the default value is 0 in this example.

   Include the standard rule used for the management of the IOM tunables in `iom_action.xml`:

```
 <setting name="iom.tunables">
   <condition><ifdef name="iom.my.tunable"></condition>
   <value index="size">
   <vstring>my_tunable iom.my.tunable ${iom.my.tunable}</vstring>
   </value>
   </setting>
```

Note - `iom.my.tunable` is the external name of the tunable and `my_tunable` is the corresponding integer declaration in the source code.

2. Update the configuration in the build directory and build the new system image. As you modified XML configuration files in the source directory, you must propagate these changes back to the work directory. Remove the corresponding XML files and run `make xml`.

```
host% cd <work_dir>
host% rm conf/mkconfig/iom_rule.xml conf/mkconfig/iom_action.xml
host% make xml
```

3. Set your tunable and check that it is now visible in the configuration:

```
host% configurator -set myiom.my.tunable=0x12345
host% configurator -list tunables | grep my
        myiom.my.tunable:'0x12345'
```

4. Build the new system image and check that the tunable is in the IOM actor:

```
host% make build
host% powerpc-elf-nm image/RAM/chorus/bin/N_iom | grep my_
a00b44f0 D my_tunable
  a00b44f0:       00 01 23 45     .long 0x12345
```

Adding a Feature

A module is source code which implements a feature. Create a file `test.c` as shown for the creation of `bye.c` in the `mkmk` tutorial in Chapter 5. The module may be created so that it will (or will not) be present in the IOM component. This depends on the value of the feature `MY_TEST`. By default, the module will not be included in the IOM component, as explained below.

To create the test module, carry out the following steps:

1. Create the `test.df` production file for the test module:

```
host% cd <src_dir>/iom/src/os/iom/sys/test
```

2. Create a `test.df` file containing:

```
MODULES=module_test
```

The module named `module_test` will contain any object files present in the directory and subdirectories. The module is a collection of object files, in this example, it will only contain `test.o`.

3. To add the feature in the XML source configuration file for the IOM component, go to the directory containing the IOM configuration files:

   host% **`cd <src_dir>/iom/conf/mkconfig`**

4. In order to add the MY_TEST feature, modify both of the `iom_rule.xml` and `iom_action.xml` files. Include the definition of the feature in XML in the `iom_rule.xml` file:

```
 <feature name="MY_TEST">

              <description> My Feature </description>
              <bool>
              <false>
            </feature>
```

   The definition includes the name of the feature, a description field and a value. Both the description field and the value will be accessible through the `ews` configuration tool. In this example, the value is a boolean value, false by default. As the default value is false, the module will not appear in the IOM component. Include the standard rule used for the management of the IOM features in the `iom_action.xml` file:

```
<setting name="iom.modules">

            <description>module_test</description>
            <condition>

              <var name="MY_TEST">
            </condition>
            <value index="size">

              <const>module_test</const>
            </value>
            </setting>
```

Note - MY_TEST is the external name of the feature, `module_test` is the corresponding module managed by the `mkmk` tool.

---

5. Rebuild the IOM component:

   host% **`make mkmk`**
   host% **`make depend`**
   host% **`make`**

6. Update the configuration in the build directory and build the new system image:

```
host% cd <work_dir>
host% rm conf/mkconfig/iom_rule.xml conf/mkconfig/iom_action.xml
host% make xml
host% make build
```

As the newly created `test` module is set to FALSE by default, you will not find the definition of my_tunable by running:

```
host% powerpc-elf-nm image/RAM/chorus/bin/N_iom | grep my_
```

In order to configure the MY_TEST feature, type:

```
host% configurator -set MY_TEST=true
```

and the test module will be integrated.

Adding a New XML File

XML files used during ChorusOS system generation are copied to the `conf` directory, so that they can be modified using the `configurator` command or the `ews` graphical tool. A component that adds new XML files must have rules to execute these files in its `Makefile.bin` file. For instance, the `Makefile.bin` of the IOM component contains:

```
IOM_XML = iom.xml iom_rule.xml iom_action.xml

xml:: DEVTOOLS.all $(IOM_DIR)/exports.lst
sh $(DEVTOOLS_DIR)/cpxml $(BUILD_DIR)/conf/mkconfig $(IOM_DIR)/conf/mkconfig $(IOM_XML)
```

In this example, the XML files have to be copied from the IOM merged tree, so the XML target depends on `$(IOM_DIR)/exports.lst` (which is produced by `mkmerge` during the IOM merge). The `cpxml` command acts as a wrapper around `cp`.

The `conf/ChorusOS.xml` file includes, directly or indirectly, all the XML files used during the ChorusOS system generation. This file is generated automatically. As the inclusion order of XML files is important, the generation of `conf/ChorusOS.xml` uses the XML0, XML1, XML2 and XML3 `make` variables in that order. In other words, files in XML0 are included in `ChorusOS.xml`, before files in XML1.

For instance, the `Makefile.bin` of the IOM component contains:

```
XML3 += mkconfig/iom.xml
```

For further information on XML files, see Chapter 7.

CHAPTER 7

XML Syntax

This Chapter gives a formal description of the XML language used in the ChorusOS configuration files.

The `conf` directory described throughout this document includes several XML files that you can use as examples when you need to modify the XML configuration files. The *ChorusOS 4.0 Porting Guide* also explains how to modify the XML configuration files when adapting ChorusOS to a new target platform.

XML Files

The grammar given here is a simplified overview of the ECML language. ECML is a language based on XML. The ECML DTD is located in `<bin_dir>/tools/ews/ChorusOS.dtd` file.

TABLE 7–1 Semantics of Grammar in XML Files

[a]	means that the a element is optional and can be omitted
{ a | b }	means that either the a element or the b element can be used
a*	means that the a element can be repeated 0 or more times
a+	means that the a element can be repeated 1 or more times

The Attributes Tables in the sections below describe the XML attributes that can be used with the tags.

Configuration

This is the topmost scope level. A configuration must start with a `<configuration>` tag.

```
<configuration>
  [ <description> ] [ <typedef>* ] [{ <definition> | <feature> | <tunable> ]*
  [ <setting>* ] [ <action>* ] [ <constraint>* ]
  [{ <folder> | <folderRef> }]*
```

TABLE 7–2 Attributes for Configuration

Attribute	Description
name: string	Name of the configuration

Folder Declaration

A folder is used to scope variables, and to set up a hierarchy in the ChorusOS system configuration. A condition can be used on a folder to disable it (if the condition evaluates to false). Any element located in a disabled folder is also disabled.

```
<folder>
  [ <description> ] [ <typedef>* ] [ <definition> | <feature> | <tunable> ]*
  [ <setting>* ] [ <action>* ] [ <constraint>* ]
  [ <folder> | <folderRef> ]*
```

TABLE 7–3 Attributes for Folder

Attribute	Description
name: string	Name of the folder
visible: ’yes’ or ’no’	Default: ’yes’. Specifies if a configuration tool should show this folder or not. If the folder is not visible, all of its content is hidden as well.

Folder Link

A folder link is used to include another file. This is used to split the configuration into several files.

```
<folderRef>
```

TABLE 7–4 Attributes for Folder Link

Attribute	Description
href: URL	Path of the file to include
visible: 'yes' or 'no'	Default: 'yes'. Specifies if the content of the included file must be visible in a configuration tool.

Description

This is used to bind a description to an element. The description text can be displayed in a configuration tool.

```
<description> text
```

Definition

This is used to declare a variable. The declaration is fully typed, and can be conditioned by a `<condition>` boolean expression. If the condition evaluates to 'false', the variable is not declared.

```
<definition>
  [ <description> ] [ <condition> ] type_content [{ expression | <value>* }]
```

TABLE 7–5 Attributes of Definition

Attribute	Description
name: string	Name of the variable to declare
configurable: 'yes' or 'no'	Default: 'no'. Specifies if the value of this variable can be changed with a configuration tool.
visible: 'yes' or 'no'	Default: 'no'. Specifies if this variable declaration must be visible in a configuration tool.
global: 'yes' or 'no'	Default: 'yes'. Specifies if this variable is scoped into the folder where it is declared.

Feature

This is used to declare a ChorusOS system feature. A feature declaration has the same semantics as a variable declaration, but it is always visible, configurable, and global.

```
<feature>
  [ <description> ] [ <condition> ] type_content [ <const> | boolean_values ]
```

TABLE 7–6 Attributes for Feature

Attribute	Description
name: string	Name of variable to declare

Tunable

This is used to declare a ChorusOS system tunable. A tunable declaration has the same semantics as a variable declaration, but it is always visible, configurable, and global.

```
<tunable>
  [ <description> ] [ <condition> ] type_content [ <const> | boolean_const ]
```

TABLE 7–7 Attributes for Tunable

Attribute	Description
name: string	Name of the variable to declare

Boolean Constants

This defines the 'true' and 'false' boolean constants.

```
boolean_const
  { <true> | <false> }
```

Type Content

This is the set of types available for variable declarations.

```
type_content
  { <bool> | <int> | <string> | <enum> | <struct> | <list> | <type> }
```

Integers

This is used to declare a variable as an integer. An integer is assigned a value of 64 bits.

```
<int>
```

TABLE 7–8 Attribute for Integer

Attribute	Description
min: integer value	Minimum value authorized for this integer.
max: integer value	Maximum value authorized for this integer.

String

This is used to declare a variable as a string.

```
<string>
```

Enumerations

This is used to declare an enumerated value. The `<const>` sub-elements define all the possible values of the enumeration.

```
<enum>
  <const>+
```

Structures

This is used to declare a structured variable.

```
<struct>
  <field>+
```

Structure Fields

This is used to describe the characteristics of a structure field.

```
<field>
  [ <description> ] type_content expression
```

TABLE 7–9 Attributes of Field

Attribute	Description
name: string	Name of the field
optional: 'yes' or 'no'	Default 'no'. Specifies if this field can be omitted when setting the initial value of a variable in its declaration.
ref-only: 'yes' or 'no'	Default 'no'. Specifies if this field contains only a references to another variable.

Lists

Used to declare a list variable. The `type_content` sub-element is used to type the element of the list.

TABLE 7–10 Attributes of List

Attribute	Description
ref-only: 'yes' or 'no'	Specifies if this list only contains references to other variables.

Boolean Expressions

Boolean expressions are written in a post-fixed notation. They are used wherever a boolean result is expected.

```
boolean_expression
  { boolean_const | <and> | <or> | <not> | <equal> | <notequal> |
    <ifdef> | <imply> }
```

Expressions

Generic expressions are an extension of boolean expressions which can produce variable reference, string, or enumeration constants.

```
expression
  { boolean_expression | <ref> | <vstring> | <const> }
```

Variable Reference

This is used to reference another variable by its name. The variable must be declared and accessible in the scope of this reference.

```
<ref>
```

TABLE 7–11 Attributes of Variable Reference

Attribute	Description
name:string	Name of the referenced variable.

Test of Variable Existence

This test returns true if the specified variable exists or has been declared.

```
<ifdef>
```

TABLE 7–12 Attribute of Variable Existence

Attribute	Description
name: string	Name of the referenced variable.

Variable value

This allows you to get the current value of a variable. The specified variable must have been declared previously.

```
<var>
```

TABLE 7–13 Attributes of Variable Value

Attribute	Description
name: string	Name of the variable.

Conditions

A condition is used to validate or invalidate an element.

```
<condition>
  boolean_expression
```

Typedef

This declares a new type. A type definition is global in the configuration, and can be used in a 'type' reference.

```
<typedef>
  [ <description> ] type_content
```

Type

This is used to reference a type definition.

```
<type>
```

TABLE 7–14 Attributes of Type

Attribute	Description
name: string	Name of the referenced type. This type must have been declared in a typeDef.
ref-only: 'yes' or 'no'	Specifies if this type denotes a reference.

Settings

This is the assignment of a variable . The named variable is assigned with the given value. The variable must already have been declared before, and the value must have a correct type.

```
<setting>
  [ <description> ] [ <condition> ] type_content { expression | <value>* }
```

TABLE 7–15 Attribute of Setting

Attribute	Description
name: string	Name of variable

Constraints

A constraint is a boolean expression used to ensure the integrity of the configuration. If a constraint has a false value, the configuration tool considers the configuration to be invalid, and not to be used to generate a ChorusOS system image.

```
<constraint>
  [ <description> ] boolean_expression
```

TABLE 7–16 Attribute of Constraint

Attribute	Description
name: string	Name of the constraint.

Actions

An action takes a parameter which specifies the precise effect that will be performed by a configuration tool.

```
<action>
  [ <description> ] [ <condition> ] <application> {<definition>|expression}
```

TABLE 7–17 Attribute of Action

Attribute	Description
name: string	Name of the action

Action Application

This determines which action of the configuration tool will be applied. Currently, only the `configure` application exists and uses a variable typed as `Configure` structure as a parameter.

```
<application>
   text
```